7th May 2015

Intelligent Agent for Open Face Chinese Poker Web-Application

Submitted May 2015 in partial fulfilment of the conditions of the award of the degree BSc (Honours) Computer Science

Alastair Kerr

axk02u

School of Computer Science and Information Technology

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature _____

Date ____/____/____

Abstract

Artificial Intelligence (AI) is a crucial and high-interest area of research in the field of Computer Science, which has gained increased traction in recent decades due largely to the requirements for increasingly sophisticated AI in games, and the implications that development of *algorithms* and *heuristics* in this area can have beyond the domain of game-playing in other fields of study such as *Automated Planning and Scheduling*. Poker games provide a challenge to *Intelligent Agents* because of many factors including *combinatorial explosions* of *game trees*, elements of *hidden information* such as the cards other players hold, as well as stochastic elements due to not knowing which cards are yet to be dealt. This differentiates Poker games from more traditional games such as Checkers, which is a deterministic *perfect information* game meaning that each player has the same complete knowledge of the game state at any stage and that there is a finite set of moves for each player. Therefore at each stage there is an *optimal move* leading to a winning strategy, and in this manner these games can effectively be *solved*, and so when playing against a competent Agent for such a game it is impossible to win, only to draw. Poker games are different because of the aforementioned stochastic elements, imperfect information and complex search trees, necessitating the use of more sophisticated algorithms in order for an Agent to perform competently versus intelligent opponents. While there has been lots of research into Intelligent Agents for traditional board games and variants of Poker such as *Texas Hold'Em*, lesser known variants such as the relatively new *Open Face Chinese Poker* have not been explored to the same degree. Agents for Open Face Chinese Poker often suffer from poor performance due to reliance on simple algorithms and methods such as *Rule-Based Systems*, which can lead to predictable or sub-optimal play that is additionally largely domain specific. This dissertation considers the merits and limitations of various AI techniques, and implements an Intelligent Agent for a bespoke Open Face Chinese Poker Web-Application, with discussions of the range of technologies used and methodologies employed in creating a functional final product. It is advisable for readers unfamiliar with Poker, Computer Science or any of the italicised terms found throughout to familiarise themselves with the definitions found in the glossary section.

Contents

1 Introduction

1.1 Outline of Introduction

Section 1.2 covers an introduction to Open Face Chinese Poker, explaining the basic rules and giving illustrated examples of the game board and the scoring system. Readers unfamiliar with the game will benefit largely from reading this section in conjunction with reference to the glossary of terms.

Section 1.3 outlines the problem description, considering the limitations of existing Intelligent Agents and the reasons for these limitations.

Section 1.4 outlines specific aims and objectives for the project based on functional, non-functional and usability requirements for function, performance and meeting user demands.

Section 1.5 discusses the potential for ethical issues in this project and its area of research.

1.2 Introduction to Open Face Chinese Poker

Open Face Chinese Poker (commonly abbreviated to *OFCP*) is a perfect or imperfect information (depending on variant)[1] card game, and is a variant of *Chinese Poker*. In OFCP, players take turns placing cards face-up into three distinct *rows* (*bottom*, *middle* and *top*), creating the best possible *poker hands* they can in order to score points from each other. Stronger hands score bonus points called *royalties* and royalty-scoring hands in higher rows score more points than equivalent hands in lower rows. For example, a *Royal House* gives +25 bonus points in bottom row, or +50 in the middle row. Players score +1 point for each row they win in addition to any royalties. In the case that a player wins all 3 rows they score a *scoop* bonus which grants an additional point for each row, for a total of +6 base points before royalties are calculated. OFCP is a zero-sum game, meaning any gains by one player are balanced with losses by another player; players win points directly from their opponents, so if a player's score is +16 points then in a 1v1 game it is therefore implied that their opponent's score would be -16.

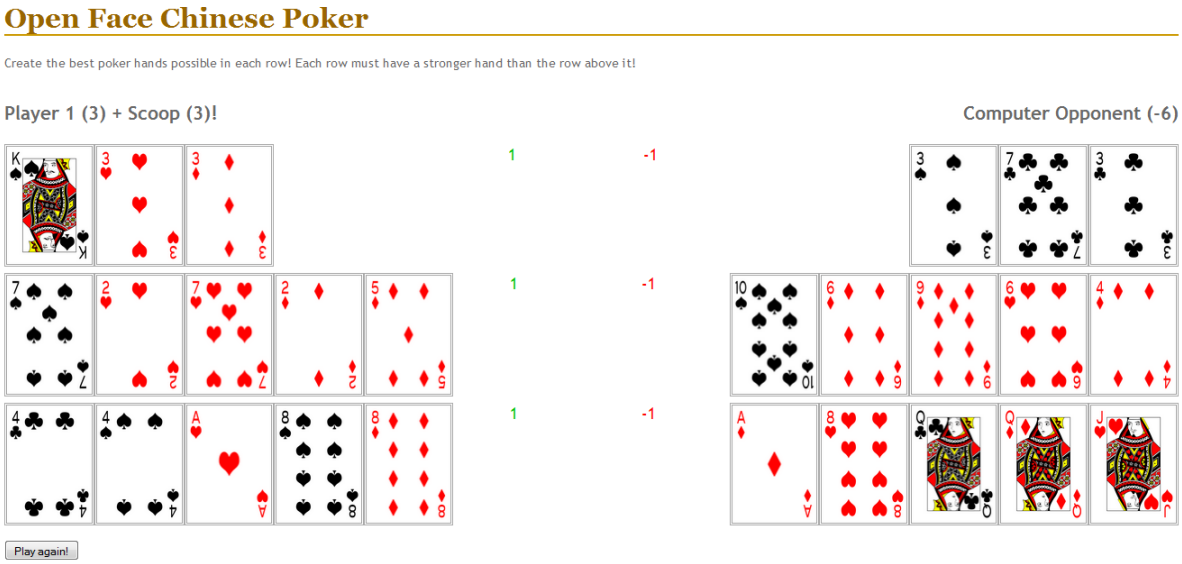


Fig 1.2.1 – layout of the board after a round of Open Face Chinese Poker (screenshot from an early prototype of the application)

Figure 1.2.1 shows the results and layout of the board after a game of OFCP. Player 1 wins bottom with a *Two Pair* 8s and 4s versus a *Pair* of Qs for +1 point. Player 1 also wins middle row with Two Pair 7s and

[1] Standard OFCP is a perfect information game, but other variants such as *Pineapple OFC* also feature elements of hidden information

2s versus Pair of 6s for +1 point. In top row Player 1 and the computer opponent both have a Pair of 3s, and so the third card is taken into account as the *kicker*, with Player 1 clinching the win for +1 point with a King kicker versus a 7. Further to the individual row scores, because Player 1 won every row they score an additional scoop bonus of +3.

One important caveat of the game is that hands in lower rows must be stronger than those on the rows above; if a player creates a top-heavy board then that player's hand is invalid, which is known as *fouling*. When a player fouls their opponents automatically scoop them for +6 each (+1 for each row and +3 scoop bonus) in addition to any royalties, and any of the fouled player's royalties are disqualified. In the case that all players foul, no points are awarded.

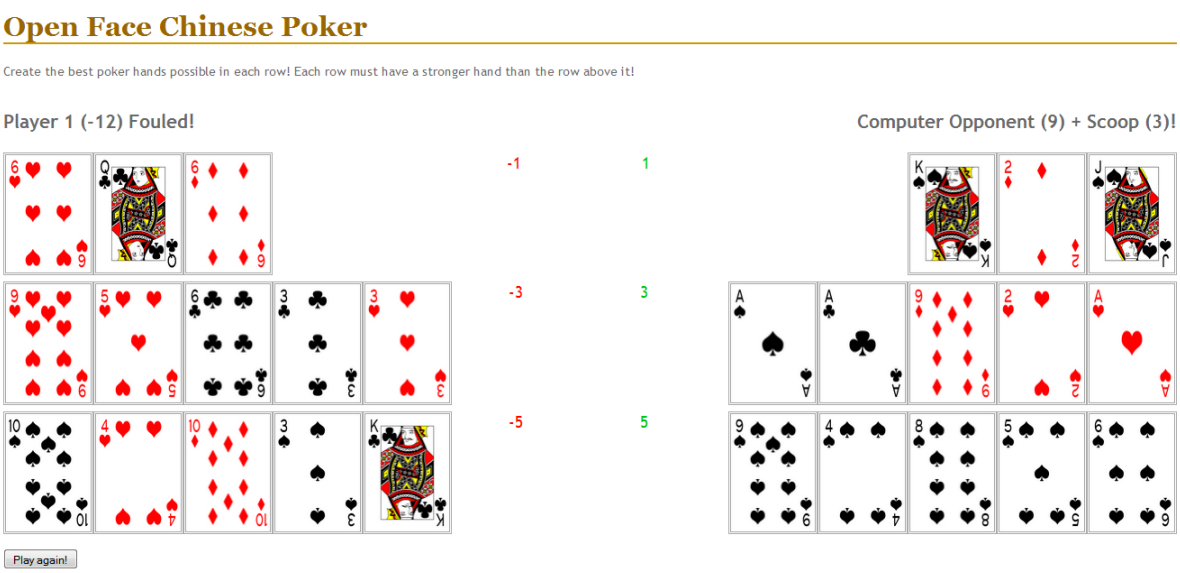


Fig 1.2.2 – Player 1 fouls and so their opponent wins an automatic scoop bonus plus royalties

In the game shown in Fig 1.2.2 Player 1 has Pair of 10s in bottom, Pair of 3s in middle and Pair of 6s in top. Because the top row contains a stronger poker hand than the row below it, the hand is invalid and the player fouls. On top of the +3 scoop bonus and points for individual row wins, the computer opponent wins further points for royalties because of its *Flush* in bottom and *Three of A Kind* in middle.

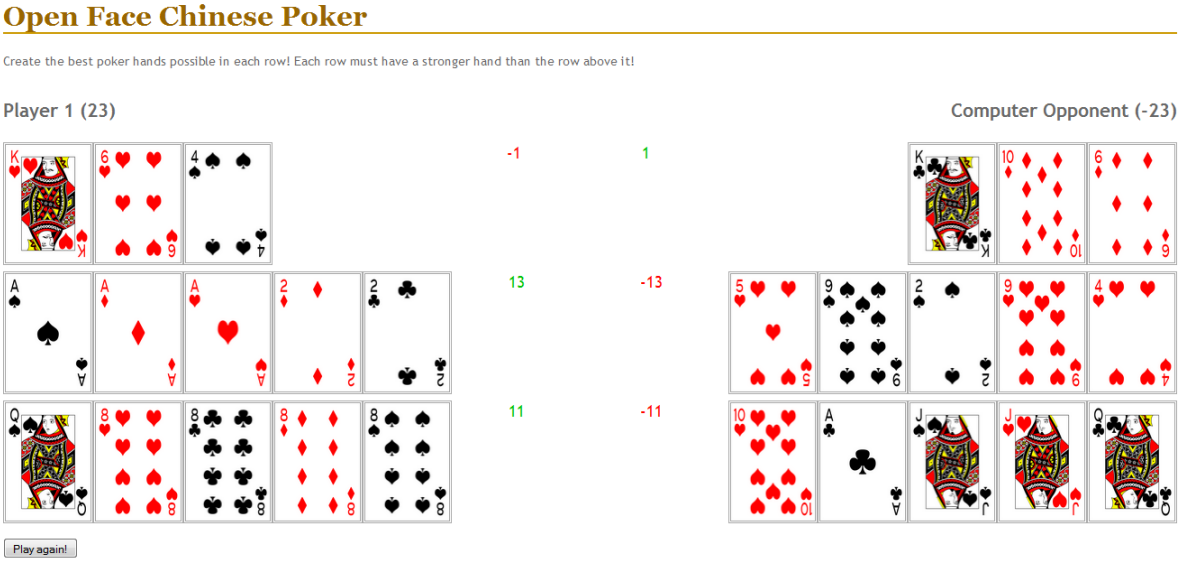


Fig 1.2.3 – Player 1 wins a lot of bonus points from royalties for strong hands

Fig 1.2.3 indicates how many points can be won from strong hands. Player 1 wins bottom row with *Four of a Kind* 8s versus the computer opponent's Pair of Jacks. Player 1 receives +1 point for winning the row, with an additional +10 points royalty for Four of a Kind on bottom. Player 1 also wins middle row with a *Full House*, Aces full of Deuces versus the computer opponent's Pair of 9s, scoring +1 point for winning the row and an additional +12 royalty. Player 1 and the computer opponent both have *High Card* King in the top row, but the computer opponent wins the row because their kicker of 10 beats Player 1's 6 kicker. The computer opponent wins +1 point for winning this row but the hand is not strong enough to score any additional royalties. The points each player wins are taken from their opponent, so the final score for Player 1 is $-1 + 13 + 11 = 23$, and the computer opponent's score is the inverse of this, -23.

1.3 Problem Description

Creating sophisticated Intelligent Agents for games with large branching factors poses a significant challenge, as for the Agent to perform well it must be able to search many moves ahead of the current state in order to find the optimal move. With every additional level of depth searched in a game tree the complexity of the search increases exponentially due to combinatorial explosion; at a depth of d with a branching factor B there would be approximately B^d potential nodes to explore. A brute-force Depth First Search would therefore have a time complexity of $O(B^d)$ which is prohibitively complex for any reasonably exhaustive search of a game with a high branching factor. Consider for example Chess, which has an estimated branching factor of 35 (Mandziuk, 2010, p4). In his influential paper *Programming a Computer for Playing Chess*, Claude Shannon estimated a lower-bound for the game-tree complexity of Chess of 10^{120}, asserting "A machine operating at the rate of one variation per second would require over 10^{90} years to calculate the first move" (Shannon, 1950, p4). While in theory exhaustively analysing a game of chess from start to finish is possible, it remains implausible with any conceivable modern computer despite huge advances in processing speed and power since 1950.

Search Depth	**Approximate Nodes**	**Approximate Time to Search (10^6 nodes/sec)**
1	35	3.5×10^{-5} seconds
2	35^2	1.2×10^{-3} seconds
3	35^3	4.3×10^{-2} seconds
4	35^4	1.5 seconds
5	35^5	52.5 seconds
6	35^6	30.6 minutes
7	35^7	17.9 hours
8	35^8	26 days
9	35^9	2.5 years
10	35^{10}	87.5 years
11	35^{11}	3061 years
12	35^{12}	$\sim 100,000$ years
20	35^{20}	$\sim 2.4 \times 10^{17}$ years
30	35^{30}	$\sim 6.7 \times 10^{32}$ years
40	35^{40}	$\sim 1.8 \times 10^{48}$ years

Table 1.3.1 showing the intractability of a brute force search of a Chess game tree

Open Face Chinese Poker does not have such an astronomical space state or game-tree complexity as Chess, although unlike Chess it incorporates stochastic elements because of its nature as a card game, and the subsequent challenge this poses for an Intelligent Agent makes it an interesting area of research. Considering there are $52! \approx 8 \times 10^{67}$permutations of a standard deck of cards, representation of each possible state in a game tree invokes a combinatorial explosion of branching factor. An upper-bound[2] for possible final

[2]This estimate is generous as it includes duplicate states e.g. rows with the exact same cards but in different orders

game states is 4.96×10^{14} with an upper-bound estimate for the game tree size of 5.77×10^{32} which poses a similarly daunting complexity. The game tree size was calculated by assuming that both player's first 5 cards are known and then taking into account from there each player's sequential turns for the duration of the game, using the following formula:

$$\left(\binom{3}{1} \cdot 42\right) \times \left(\binom{3}{1} \cdot 41\right) \times \cdots \times \left(\binom{3}{1} \cdot 30\right) \times \left(\binom{2}{1} \cdot 29\right) \times \left(\binom{2}{1} \cdot 28\right) \times \left(\binom{1}{1} \cdot 27\right) \times \left(\binom{1}{1} \cdot 26\right)$$

Search Depth	**Approximate Nodes**	**Approximate Time to Search (10^6 nodes/sec)**
1	1.55×10^4	1.5×10^{-2} seconds
2	2.18×10^8	3.6 minutes
3	2.75×10^{12}	31.8 days
4	3.12×10^{16}	9.9×10^2 years
5	3.15×10^{20}	9.99×10^6 years
6	2.81×10^{24}	8.9×10^{10} years
7	1.47×10^{28}	4.65×10^{14} years
8	2.22×10^{31}	0.7×10^{18} years

Table 1.3.2 showing the possible states to consider in an exhaustive search of an Open Face Chinese Poker game tree

The rapid expansion of the search space makes it clear that more processing power is not in itself a solution to the problem, as combinatorial explosion quickly makes deep searches of these games impractical for any realistic purposes; even a machine that could search 1 Trillion (10^{12}) nodes per second would take 7×10^{11} years to perform such an exhaustive search of an entire OFCP game tree.

However, despite the fact that modern computers are still not powerful enough to perform such complex and exhaustive searches, competent Intelligent Agents for games such as Chess have emerged regardless. The reason this is possible is because of different approaches to the problem, or through pruning algorithms which reduce the complexity of the search by removing branches of the tree that do not provide beneficial information. A variety of algorithms and heuristics have been formalised over the decades, with the creation of a multitude of Intelligent Agents of varying competence, with wide coverage of more traditional games such as Chess and Checkers, and even for more complex games such as Go. In terms of artificial intelligence applications for Poker, more well-known variants such as Texas Hold'Em have seen lots of interest while more obscure variants such as OFCP have remained somewhat untouched.

Intelligent Agents do exist for OFCP, but publicly available implementations largely play predictably or sub-optimally, often favouring conservative play over taking risks, even in situations where fouling would not be heavily penalised (such as when the human opponent is in a fouling position themselves). There has been little formal research into Intelligent Agents for OFCP, and as such limited source material is available. Most existing bots for the game are designed for Poker sites which do not have any source code made available, or made by development teams which license the software out to Poker players for money such as in the case of Warren[3]. Even with such advanced techniques the game has not been solved, as evidenced by the modest claim that "(Warren) plays almost perfect Open Face Chinese Poker after the 6th card". Other implementations of Agents for OFCP by hobbyist programmers for personal projects often make use of simple algorithms which leads to sub-optimal, exploitable play because of predictable or naïve behaviour. This project aims to create an Intelligent Agent that performs well versus competent human players, and to do so will need to be able to do everything a human can in terms of making judgement calls, exploiting the player's game position in order to inform its decision as to whether it should take a risk or play conservatively to avoid fouling itself.

[3]Warren is an OFCP bot which uses neural networks to train and improve, licensed out for use at a cost of between $7-79/month depending on chosen package (https://www.playwarren.com/)

1.4 Aims and Objectives

The project aim is to create a functional, competent Intelligent Agent for a bespoke Open Face Chinese Poker Web-Application, which can be broken down simply into two main objectives.

1. Create a game environment for Open Face Chinese Poker (Web-Application)
2. Create an Intelligent Agent which interfaces with the Application and plays the game

Beyond these simple functional requirements the Intelligent Agent must meet expectations for competent performance, making strong plays and avoiding the pitfalls of more simplistic Agents. To create a credible degree of perceived intelligence, the Agent must play tactically, avoiding fouling on one hand but not playing in such a risk averse and conservative manner as to limit potential for scoring and exploitation of a weak player position. In addition to this, the Agent must be able to make its decisions quickly in order to minimise delay in line with user demands for responsiveness.

With regards to the Web-Application itself there are multiple demands to be met in terms of reliability, usability and functionality. The interface must be intuitive and aesthetically appealing, and must feel responsive and efficient. As it will be a Web-Application accessible via the internet efforts must be made to optimise performance and decrease loads times through means such as compression of assets and redirection minimisation, also enabling larger support for multiple concurrent users and increasing scalability.

1.5 Ethics

Ethical issues can be a concern in Poker games when playing for money; consider for example the implications of a human player unknowingly playing versus a Poker Bot. This is certainly a major concern for variants of Poker such as Texas Hold'Em which has a large online following with numerous websites and applications where virtual Poker games are played for real money. Open Face Chinese Poker on the other hand is generally only played for money in home games, with friends or at casinos as a side-game during or after a Texas Hold'Em tournament. While some websites do exist for playing OFCP for money, these do not have the same level of following as more popular variants of Poker, and the Intelligent Agent produced in this project will not provide any kind of support for integration with these websites.

In the context of the proposed application, no money will be involved and participants will be playing purely for entertainment value and research interest in Artificial Intelligence. This mitigates potential ethical issues as the project focuses not on aspects of gambling but rather on the technologies and methodologies used in creating the Web-Application and system architecture, as well as the pros and cons of various algorithms for producing a sophisticated Intelligent Agent.

2 Background

2.1 Game Theory

In game theory, a zero-sum perfect information game such as Chess or Checkers can be formalised as a search problem, representing possible game states in a game tree (Özcan, 2004, p282). Such a tree can be explored and analysed in order to determine the best move from a given position, and to do so it is necessary to implement some form of search algorithm.

2.1.1 Minimax

While game theory has been discussed at least as far back as 1713, it wasn't until John von Neumman's research into the field that game theory was established as a mathematical discipline. Neumann's early efforts focused on proving the Minimax Theorem, which was published in his 1928 paper *Zur Theorie der Gesellschaftsspiele* (The Theory of Board Games). Proving the Minimax Theorem was considered by Neumman to be of utmost importance, saying "I thought there was nothing worth publishing until the Minimax Theorem was proved" (Casti, 1996, p19). The Minimax Theorem states that in any finite zero-sum two player game

there exists strategies for each player that minimise their maximum losses. These strategies must consider their adversary's possible responses, and the strategy which minimises a player's maximum losses is known as the optimal strategy. Expanding on his 1928 paper, Neumman co-authored *Theory of Games and Economic Behavior* in 1944, which presented a different proof of the Minimax Theorem, extending the theorem to include imperfect information games and games with more than two players. This 1944 publication was the first formal, coherent book in the field of game theory and is the basis for modern game theory.

Minimax searches a game tree, looking for moves that maximise a player's chance of winning. A minimax approach uses an evaluation function to score possible game states, choosing the moves that lead to the highest expected value for a given Agent. Implementation of a minimax search assumes that the opponent will play optimally and thus choose the next state with the lowest score from the Agent's perspective. A naïve minimax approach will produce optimal results assuming a perfect evaluation function, but explores nodes unnecessarily and takes a very long time because it will exhaustively expand the game tree, and due to combinatorial explosion is prohibitively slow for all but the most simple of games, or when exploration of the tree is limited to a certain depth. Tic-Tac-Toe for example has $9! = 362,880$ possible outcomes, which is feasible for a computer to compute, but this huge amount of nodes to consider for such a simple game demonstrates the impracticality of this naïve approach. For these reasons its use is limited in games with high branching factors unless optimisations are made.

2.1.2 Alpha-Beta Pruning

Alpha-Beta Pruning is an optimisation of the minimax algorithm, reducing the number of nodes that must be visited in a search. It was discovered independently a number of times by various researchers throughout the mid 20th century, and Arthur Samuel implemented an early version for Checkers in 1959 (Samuel, 1959, p210-229). This optimisation of the minimax algorithm is achieved through keeping track of and updating two scores; Alpha, which keeps a record of the best score possible by any means; and Beta, the worst-case scenario for the opponent. Any move with a value lower than this Alpha score can safely be pruned since a better move has already been found, and with the Beta value it can be assumed that any discovered move with a value higher than this will not be used by the opponent. With Alpha-Beta Pruning, a branch of the game tree is pruned when at least one possibility is found that is worse than a previously examined move. Pruning away branches in this way does not affect the quality of the result as these branches could not possibly influence the final decision, and therefore Alpha-Beta Pruning returns the exact same move as a standard minimax approach would, but does so faster and more efficiently. In the worst case Alpha-Beta Pruning must examine $O(b^d)$ leaf nodes, although assuming that best moves are always searched first, the number of leaf nodes evaluated will be as low as $O(\sqrt{b^d})$, reducing the branching factor to its square root, effectively meaning a search can be made to twice the depth with the same amount of computation (Russell, 2003).

2.2 Solving Games

Even with optimised approaches to game tree searches, due to high average branching factors managing and mitigating the complexity of game trees remains a challenge. Simple games such as Tic-Tac-Toe can be trivially solved, and even more complex games like Checkers can be weakly solved with traditional algorithms. However, solving more complex games such as Chess and Go still remains a challenge. Chess can be partially solved with retrograde algorithms, solved for only some positions or when using smaller non-standard boards. Solving Chess is generally considered impossible with modern technology, and it is debatable as to whether future improves in computer processing speeds will one day allow for brute-force solving of the game. Intelligent Agents for Go generally can only perform well in a limited capacity, such as when playing on a much smaller than usual board. It is not expected for full-size 19x19 board Go to be solved anywhere in the immediate future, although modern approaches to the problem have produced somewhat promising progress in terms of performance.

Game	State-Space Complexity	Solved	Method
Tic-Tac-Toe	10^3	Strongly	Traditional Algorithms
Checkers	10^{20}	Weakly	Traditional Algorithms
Chess	10^{47}	Partially	Retrograde Analysis
Go 19x19	10^{171}	-	-

Table 2.2.1 - Showing the current state of some popular board games

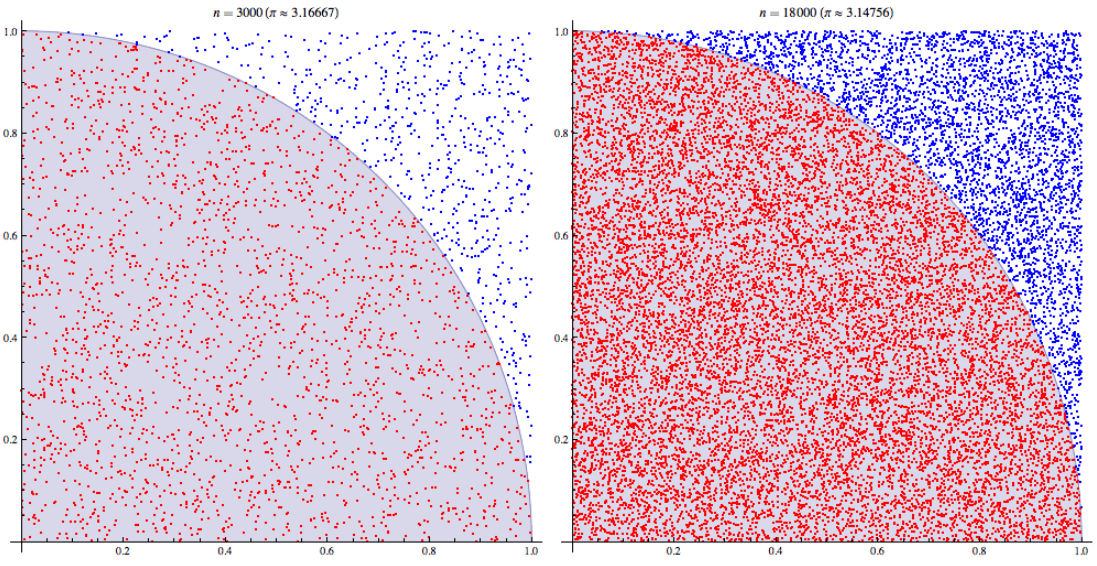
2.3 Games of Chance and Imperfect Information

The aforementioned board games have all been zero-sum perfect information deterministic games, which makes them well suited for the application of traditional search methods such as minimax and its derivatives. However, games such as Poker are stochastic in nature because of random cards, and in addition to any elements of imperfect information, this presents a challenge for implementation of an Intelligent Agent. Imperfect Information game trees can be made, using chance and decision nodes that are grouped into information sets, however since these nodes are not independent it means that algorithms such as Alpha-Beta Pruning cannot be used (Dizman, n.d, p3).

2.4 Monte Carlo

Monte Carlo methods are computational algorithms which use random sampling over multiple trial runs in order to obtain usable, numerical results. Modern Monte Carlo methods have origins in the 1940s during work on nuclear weapons projects, and were used in the simulations required for the Manhattan Project, although were limited by the lack of technology at the time (Eckhardt, 1987, p131).

Implementation of Monte Carlo methods generally involves defining possible inputs and then randomly generating inputs within this range, computing the value or result for each simulation and finally aggregating results. An example of this process is illustrated below in the context of calculating π by random distribution of points within a given area. As the number of trials increases the fraction of points distributed within the circle's area diverges on $\frac{\pi}{4}$, which can then be multiplied by 4 to give a reasonably accurate estimation of π.



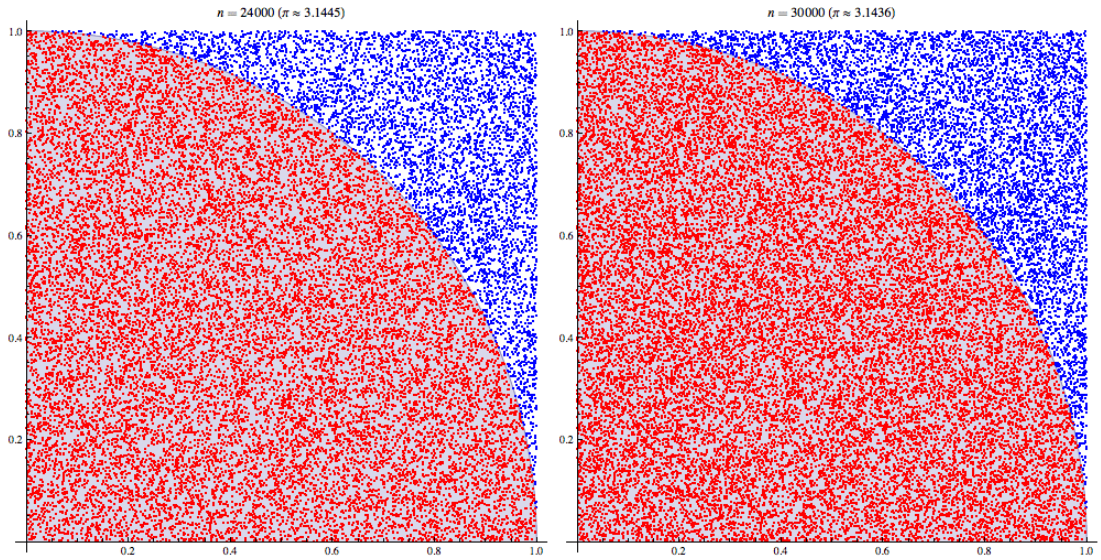


Fig 2.4.1 - Monte Carlo used in estimating the value of pi

Images licensed under Creative Commons 3.0, Author: CaitlinJo

Monte Carlo Methods have applications in a wide range of fields from physics and mathematics to business and finance, and can be particularly useful in applications where domain specific knowledge is limited or difficult to implement.

2.5 Monte Carlo Tree Search

Monte Carlo methods can be used with good results for applications in games, and since 2006 have been developed into a technique called *Monte Carlo Tree Search (MCTS)*, a term coined by Rémi Coulom during his application of the method to play Go through expansion of the search tree based on random sampling (Coulom, 2007, p72-83). In MCTS, simulated games or *playouts* are played out to the end through random move selection, and the final result of each simulation is used to weight the nodes used, meaning that better nodes are more likely to be chosen in future playouts. MCTS in its most basic form follows a fairly simply process, generating and scoring a search tree node by node according to the results of the simulated playouts. The diagram below illustrates this.

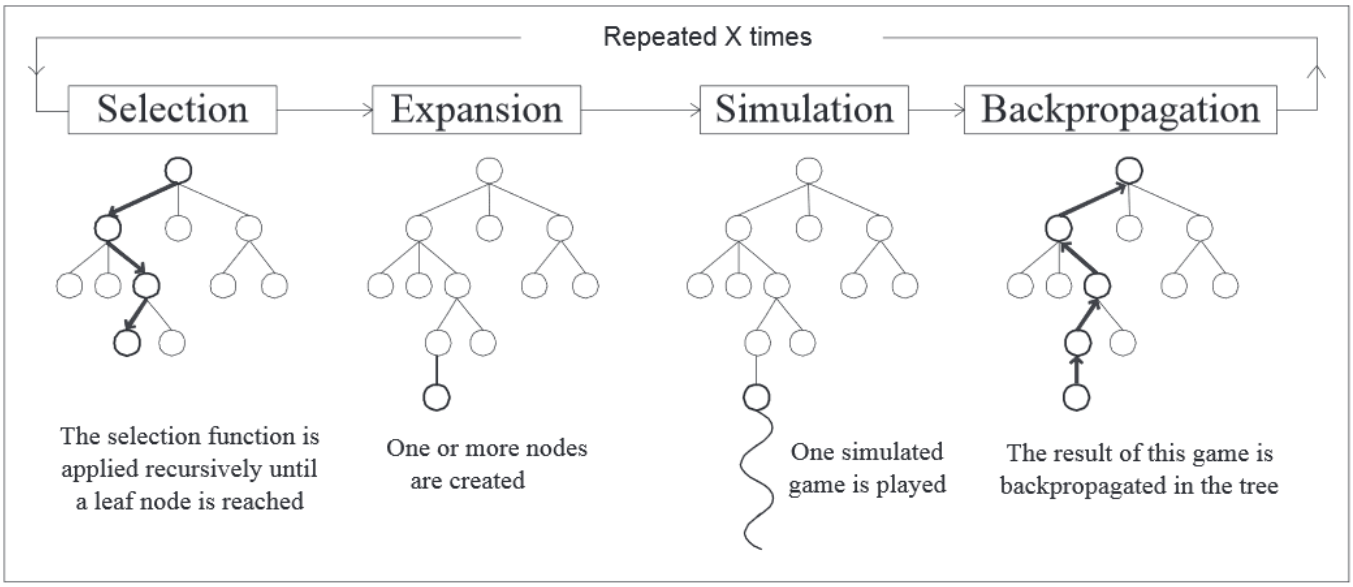


Fig 2.5.1 - Methodology of a Monte Carlo Tree Search (Figure from Chaslot et al., 2008)

MCTS has quickly gained traction as a strong general purpose algorithm for AI in games due to its effective results with potentially low space and time complexity. The game tree in MCTS grows asymmetrically, with more promising branches being favoured for exploration. Because of this MCTS can be more efficient than traditional algorithms and produce better results as it is better suited to dealing with the problems posed by high branching factors. One of the most enticing benefits of MCTS is that it requires no strategic or tactical knowledge about a problem domain other than end conditions and legal moves, making MCTS implementations flexible and applicable to a variety of problem domains with little modification (Browne et al., 2012, p9).

2.5.1 Upper Confidence Bounds Enhancement

While the basic MCTS algorithm is not perfect, there have been many proposed enhancements. Perhaps the most well known of these uses an *Upper Confidence Bounds* formula formalised by Kocsis and Szepesvári (2006, p1-12), as shown below.

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

v_i is the estimated value of a node, C is a tunable exploration parameter, n_i is the number of times the nodes has been visited and N is the total number of times the node's parent has been visited. This enhancement proves a strong balance between exploitation of high scoring nodes and exploration of nodes that have not been explored thoroughly. As with any method based on random sampling, initial estimates will not be reliable but over time and with more iterations the results will diverge on finding the optimal move. Use of UCB with MCTS is known as the *Upper Confidence Bounds for Trees* (UCT) method.

Because of the asymmetric tree growth MCTS is suitable for deep and broad explorations of larger search spaces that would be infeasible for a typical search algorithm, and because of the adaptability in regards to exploitation and exploration, optimal moves will be found and focused on. MCTS therefore has revolutionised Artificial Intelligence approaches, most notably in Go which has seen various successful applications of MCTS and MCTS with UCT to create sophisticated Intelligent Agents which present a credible challenge to even skilled human opponents[4].

2.5.2 Information Set Monte Carlo Tree Search

Information Set Monte Carlo Tree Search (ISMCTS) follows on from application of MCTS to games involving hidden information, aiming to overcome the shortcoming of previous approaches to dealing with hidden information such as determinisation techniques. ISMCTS uses randomisations of the current game state to make guesses of hidden information, with each determinisation conceivably representing what could actually be the current game state based on the Agent's observations so far (Whitehouse et al, 2013, p101).

ISMTCS is useful for games like traditional Texas Hold'Em poker where elements of hidden information mean an Agent lacks all necessary information to make a well-informed decision. ISMTCS models various possible game state variations of what other players could have, attempting to guess other players cards based on previous information. An advantage of ISMCTS over approaches such as determinised UCT is the use of better strategies which more closely resemble the decision making processes that a human player would make (Cowling et al, 2012, p121).

This approach is not needed for standard OFCP because it is a perfect information game. However, ISMCTS could come into play in an implementation of a custom variant of OFCP such as Pineapple OFC, which does have elements of hidden information due to players secretly discarding one of their dealt cards each hand. ISMCTS could be used to model which cards are unlikely to have appeared based on how the opponent plays; for example, if a player has a King on bottom row but does not pair it in the next hand then it is almost certain that the discarded card is not a King. Information can be built in this way to influence determined probabilities of certain cards appearing and the Intelligent Agent can act appropriately, tweaking expected probabilities for outs and changing strategy to match this information.

[4] 'Human-Computer Go Challenges' http://www.computer-go.info/h-c/index.html (Accessed 2015)

2.6 Artificial Intelligence in Poker

Pot limit poker solved. Not true for holdem. Where does OFC stand? Reasonable complexity for standard OFC, but other variants of OFC are even more complex e.g. Pineapple OFC.

permutations for standard OFCP game state: deck of 52 cards, each player places 13 cards (26 total for a heads up game)

52 choose 26 = 495,918,532,948,104 (4.9591853e+14) [including duplicates]

Methods e,g, database look ups impractical to implement due to space complexity of game. Need a method which has a suitable compromise between time and space complexity. Monte Carlo methods are perfect for this, especially considering the usage of heuristics which can optimise the algorithm e.g. UCT, pruning tree branches

http://scrambledeggsontoast.github.io/2014/06/26/artificial-intelligence-ofcp/ - Haskell AI for OFC 'Kachushi'. Carries out monte carlo simulations of rest of game to inform expected value for decisions.

2.7 Hand Evaluation Algorithms

For any Poker game - and indeed for any Poker AI - hand evaluator functions are necessary to rank, score and compare hands. A naïve approach to the problem could involve step-by-step comparison of a given hand against each entry in a comprehensive list of hands and matching ranks. Such a technique would generally prove efficient in terms of time-complexity and could prove to be significantly faster than performing scoring calculations on demand. One problem with this approach however is that it would necessarily involve pre-computation and scoring of every single possible hand combination, which would undoubtedly be a lengthy process. Another issue is potential memory and space constraints, dependent on the system and application in question. There are $\binom{52}{5} = 2,598,960$ possible Poker Hands, and representation of each would amount to a sizeable lookup table[5]. This could prove problematic for example in the case of a pre-packaged application; it would be incredibly undesirable to include such a large file.

In situations where such a technique is infeasible or undesirable, for example due to memory constraints, other approaches are possible. On-demand hand evaluation can be achieved in a variety of ways, for example through a simple histogram approach which reads in a hand and records the frequency of each card rank. Using this information the evaluator would be able to work out the hands rank based on highest rank frequencies. For example, a hand consisting of "2 of diamonds, 2 of hearts, 5 of clubs, 7 of spades, King of diamonds" would be represented in a histogram as seen below.

Card Rank	2	3	4	5	6	7	8	9	10	J	Q	K	A
Rank Frequency	2	0	0	1	0	1	0	0	0	0	0	1	0

Table 2.3.1 - Demonstrating how a histogram approach would store information about a hand

In this example the evaluator would then analyse the histogram, recognising that the highest frequency rank was 2 (indicating a pair), and that no other frequency was higher than 1 (indicating there is no second pair), and that the highest ranked card with a frequency above 0 was a King. A potential format of this information could be along the lines of $[1, 2, 13]$, where the first number indicates the overall hand rank (1 signifying a Pair), the second number indicating the rank of the card with this frequency (2), and the third number indicating the kicker (13 signifying a King). This could then be used directly elsewhere for comparison, or interpreted into a more readable format such as "Pair of 2s, King kicker".

The advantage of this approach comes in its simplicity as it is trivial to implement and easy to understand, and provides a strong compromise between space and time complexity. One limitation to note is that simply mapping frequencies of hands is not enough to produce complete results; additional consideration must be made to check for straights and flushes. However, this does not pose a significant challenge, as simple checks can be made if the highest frequency found is 1 to see whether all the suits are the same, or if the 5 cards are sequential.

[5] For example, this Poker hand lookup table is 100MB (https://github.com/chenosaurus/poker-evaluator/blob/master/data/HandRanks.dat)

These kind of approaches to hand evaluation have reasonable efficiency, with the potential to evaluate hundreds of thousands of poker hands per second, and considering the relative ease of implementation are perfect for most purposes. However, more efficient algorithms require less processing power enabling higher throughput and faster execution times, and as such can be hugely beneficial for large scale applications. Other approaches to hand evaluators generally are not so simple, using shrewd techniques and taking full advantage of the efficiency of low level languages such as C. One such evaluator is Cactus Kev's[6], the principle component of which is the realisation that while there are 2,598,960 possible combinations of Poker Hands, these can be sorted into 7462 distinct categories. Through the use of various methods including efficient card representations, bitwise operations, lookup tables and binary searches, an incredibly fast hand evaluator can be constructed. The main disadvantage of using such an approach is the complexity to implement, and the ease of deployment is entirely dependent on the specific environment due to the nature of compiled C code. In addition, the feasibility of using such an evaluator is dependent on the compatibility of any other technologies and languages used.

3 Design and Approach

3.1 Requirements Specification

The requirements for this project were driven by the need to meet user demands and expectations for the Application, as well as ensuring the quality of the Intelligent Agent produced. The functional requirements below describe what the system will do.

1. Create a game environment for Open Face Chinese Poker (Web-Application)
   (a) The Application must implement appropriate rules for the game - fouling, scoring system
   (b) Enable alternate round support - player acts first in Round 1, Intelligent Agent acts first in Round 2 etc.
   (c) Use appropriate networking technology to support concurrent connections so multiple users can play at once
2. Create an Intelligent Agent to play Open Face Chinese Poker
   (a) The Intelligent Agent must interface with the Web-Application in order to make its moves
   (b) The Intelligent Agent must adhere to all rules of the game
   (c) The Intelligent Agent must not use any information a human player in its position would not have (i.e. it will not cheat)

Building on these are non-functional requirements specifying how the system will work, which fall into 4 categories: Reliability, Usability, Supportability and Performance.

- **Reliability**
  1. The System must be reliable with minimal errors and failures: Long Mean-Time-To-Failure
  2. Uptime of the Application should be maximised, meaning minimal downtime for upgrades and avoidance of fatal crashes
  3. Implement support for performance monitoring through logs of errors, server actions and statistics (e.g. time taken for requests)
- **Usability**
  1. The Application should be responsive and quick to load
     (a) The website should load in an acceptable time (<1s is optimal)

[6] For more information on how this algorithm works see here (http://www.suffecool.net/poker/evaluator.html)

(b) The Application should feel responsive e.g. when dragging cards

2. Interface elements should be intuitive and easy to understand (clearly labelled buttons, instructions)
3. The aesthetics of the Application should be pleasing - good layout, colour scheme, graphical elements

- **Supportability**

1. Maximise potential for scalability through maximised throughput and appropriate optimisations to reduce Application's footprint
2. Reconfiguring the Application should not require extensive or problematic changes.
   (a) Tweaking values should be easy e.g. time limit for Intelligent Agent's decisions
   (b) Adding features shouldn't break the system - achieved through good design structure
3. The Application should be compatible with as many devices as possible
4. Installation of the Application should be easy, requiring minimal reconfiguration

- **Performance**

1. Users should not have to wait more than 5 seconds after their input for the Intelligent Agent to make its move
   (a) Employ heuristics and efficient algorithms to optimise performance
2. Stress Requirements: The Application must be able to support at least 100 concurrent users
3. Throughput: At least 95% of the time, the Application should take no more than 5 seconds for any request
4. The Intelligent Agent must play competently and provide a significant level of challenge to human opponent

3.2 Use Case and Data Flow Diagrams

Cheng is a gambling degenerate who wishes to improve his OFC gameplay

Most of the complexity of the Application will be hidden from the user, with heavy processing of data and algorithms handled behind the scenes. In terms of interaction with the application there will be a frontend interface which will allow the user to interact with and play against the Intelligent Agent. At its highest level of abstraction from the user's perspective, a use case can simply boil down to the diagram below.

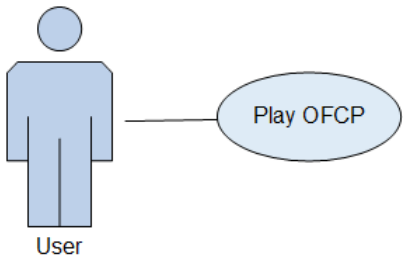


Fig 3.2.1 - Highly abstract Use Case Diagram for the Application from an end user's perspective

A more useful analysis of how a user will interact with the Application considers individual actions a user will need to be able to carry out, which is presented below.

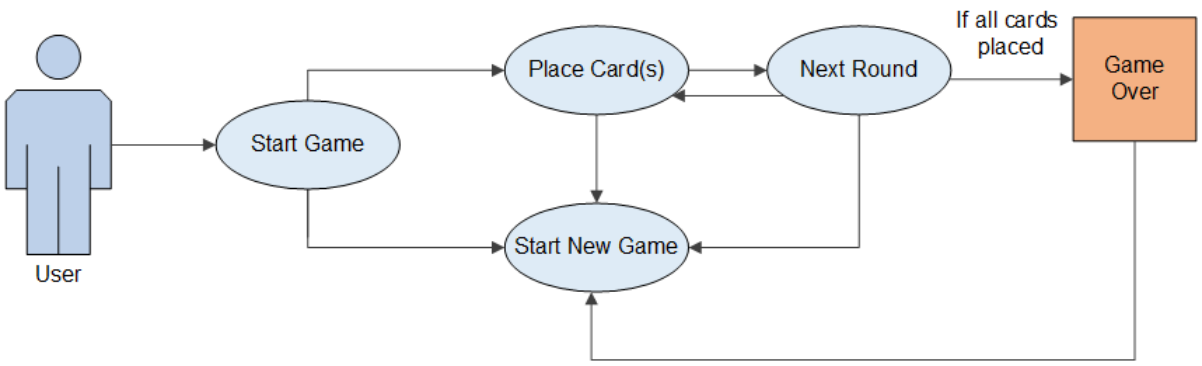


Fig 3.2.2 - Individual actions and events necessary for the user to interact with the Application

For the Application to function there must be a sequential process of user inputs followed by Application outputs, demanding further inputs and so on. For example, after a user places their cards for one round, the Application must output a response with the Intelligent Agent's card placements as well as the next card for the player to place. In order for this to work there must be at least 2 tiers for the application: a frontend and a backend. The frontend will be responsible for displaying information to the user and handling the interface, while the backend will be responsible for the main processing tasks such as processing the game state, determining the Intelligent Agent's moves and handling the game logic. This will require a 2 way flow of information from the frontend of the Application to the backend, with the frontend sending data describing the user's inputs to the backend, which will process this and return a formatted response which the frontend can interpret and display to the user.

In practice, the Application will make use of a standard three tier architecture, with a frontend for client interacton; an Application layer for the processing tasks and communication between the higher and lower tiers; and a database for storage and management of game state information. A data flow diagram has been constructed to visualise this process.

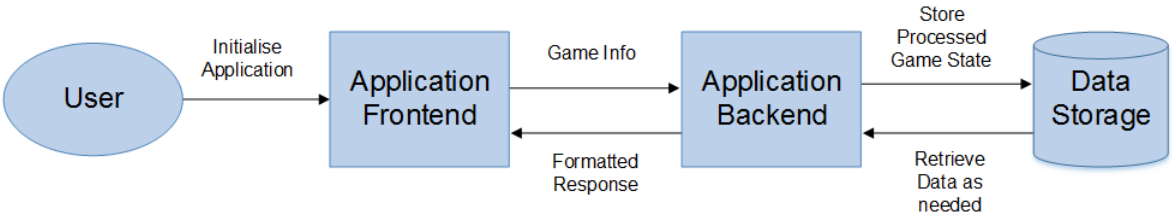


Fig 3.2.3 - Data Flow Diagram modelling the 2 way flow of information in the Application

3.3 Design Overview

Most client requests will be handled by a Web Server serving static content such as images, CSS and JavaScript files. Dynamic requests, however, will be passed by the Web Server to an Application server, which will then process the request, undertaking any necessary interactions with other pieces of software before formatting and returning a response. This response will then be passed up from the Application Server to the Web Server and from there back to the client's browser. The diagram below illustrates this process.

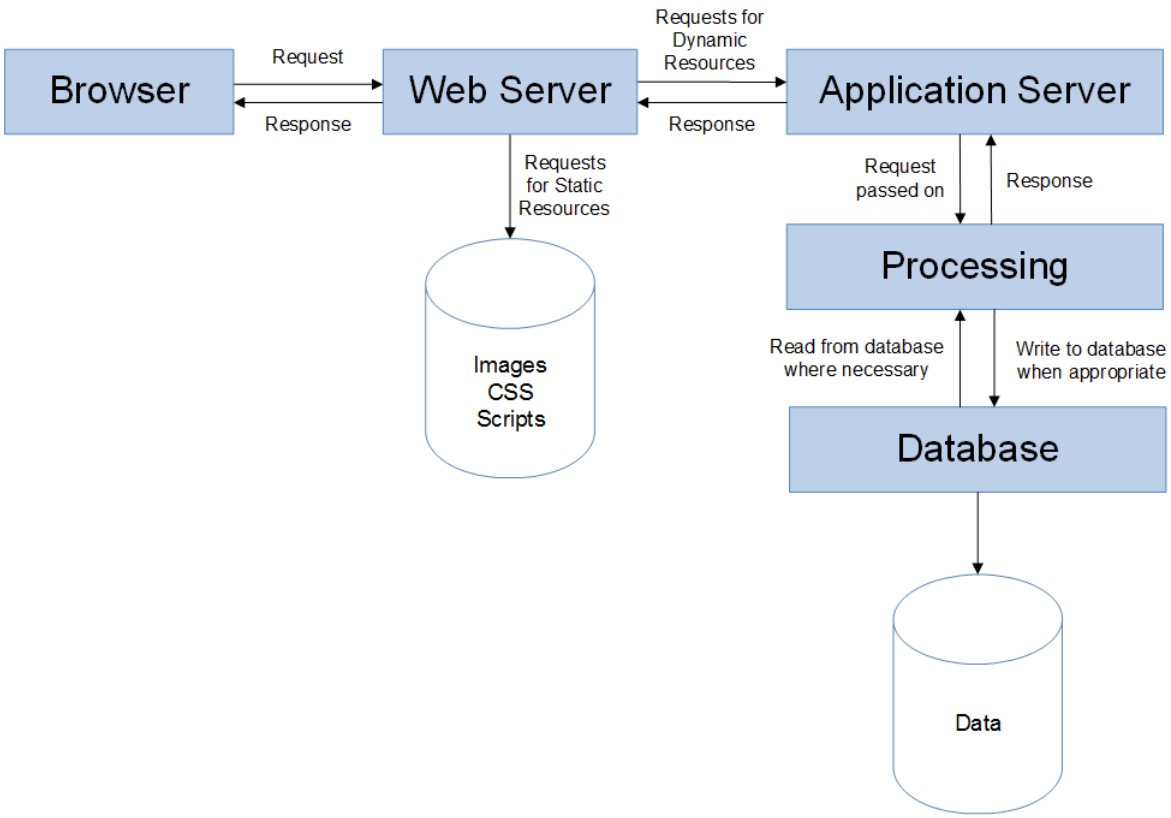


Fig 3.3.1 - System Architecture Diagram

A lower level of abstraction from this System Architecture considers specific technologies which will enable the system to function as a whole. The Application will be hosted using a Virtual Private Server running Ubuntu 14.04, aided by various technologies. Use of HTML, CSS and JavaScript will be largely responsible for the frontend of the application and will be served using an Nginx reverse proxy server. In order to produce dynamic web pages Jinja2 templates will be utilised, which will be served through Nginx by an Application Server implemented with CherryPy, a Pythonic Web Application Framework. User actions which require a response will be handled using JavaScript which will send HTTP POST requests containing the game state information to the CherryPy server. Here, the request will be parsed and sent on to the backend for processing, which will be handled with Python scripts. The backend will validate the sent game state and call helper functions and scripts to handle processes such as hand evaluation and scoring, handling the Intelligent Agent's turn and dealing cards. If everything is in order and once all necessary processes have been completed, the game state will be stored in a MongoDB document-oriented database, and an appropriate response containing information for the frontend will be produced in JSON format and sent back to the client. The JavaScript will parse and handle this response, reflecting the updates to the game state on the web page by dynamically creating and updating HTML DOM Element Objects.

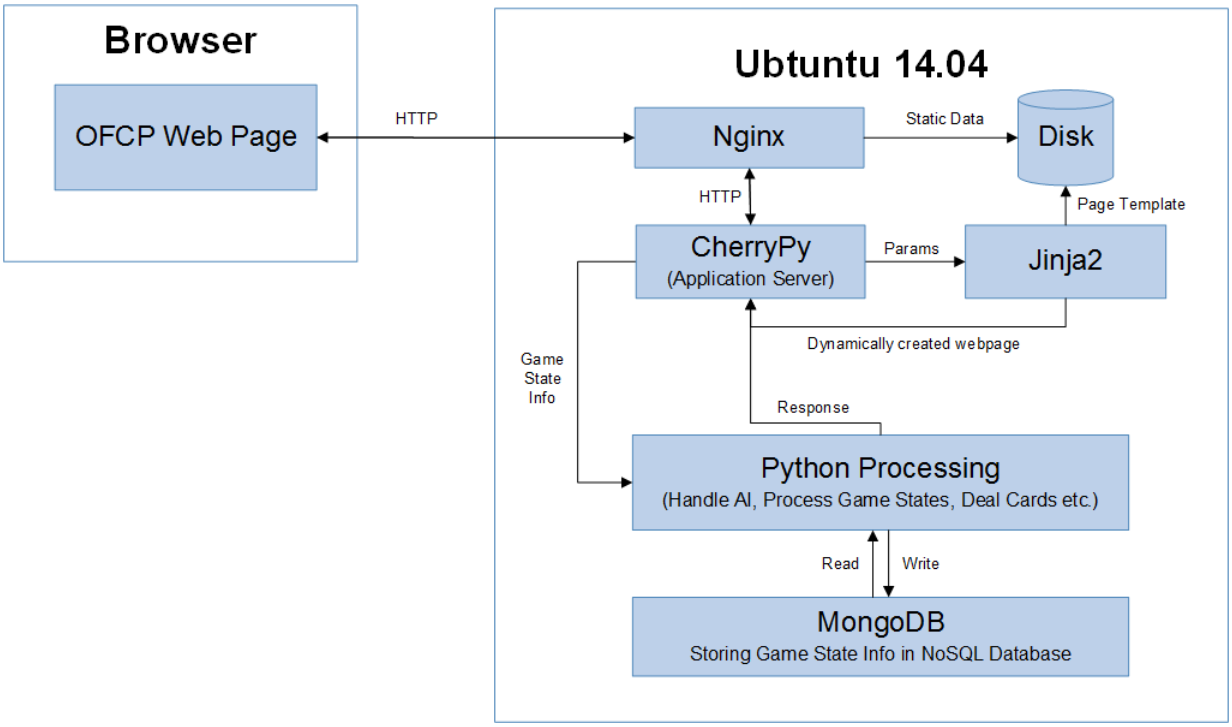


Fig 3.3.2 - Software Architecture Diagram

3.4 Wireframe

The Application's frontend will consist of one dynamic webpage, using Jinja2 templates to enable persistent information across multiple rounds and page refreshes, with content dynamically updated and created using JavaScript as required. Following Jakob Nielsen's general principles for interaction design (Nielsen, 1994, p115), the interface for the site will be made as intuitive as possible with a clear, well-structured layout. The simpler the interface, the easier to understand and use, and following this concept the wireframe mockup presented below was produced.

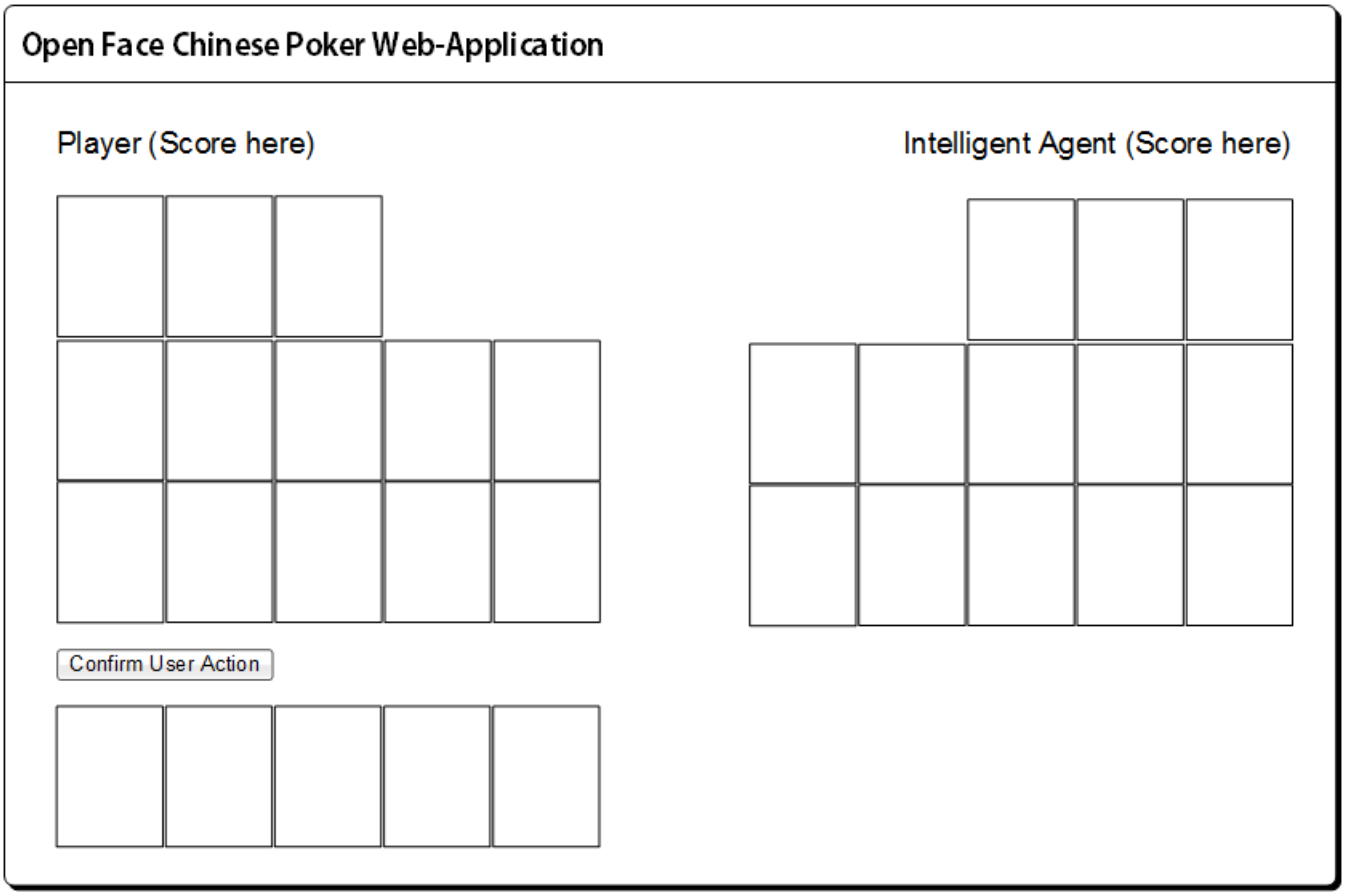


Fig 3.4.1 - Wireframe of the Interface demonstrating a clear, simple layout in adherence to principles for interaction design

The player and Intelligent Agent have distinct, separate play areas on either side of the screen where they will place their respective cards. Use of the interface is as simple as possible, with cards to be placed appearing at the bottom of the screen to be dragged and dropped into free positions on the board. The process of starting a game, finalising card placements and moving onto a new round will all be controlled through one dynamic button, making the game process as simple and intuitive as possible for the user; there are no surprises and behaviour remains consistent throughout the game. This is in strict adherence to Nielsen's aforementioned usability principles, specifically regarding consistency.

3.5 Design Principles

The design principle to adhere to throughout the process of creating the application will be Rapid Application Development using Evolutionary Prototyping, creating a functional prototype at each stage which can be refined and updated to implement new features and meet changing requirements per user feedback. This approach is ideal for the needs of the project as it allows for a flexible approach and means that emphasis can be put on development, creating a functional or semi-functional application at each stage, implementing some of the planned features, meeting some of the requirements and being ready to build upon and develop further into a new version which improves upon itself. This flexible style is naturally advantageous over a more traditional approach such as the Waterfall model which involves rigorously defining specifications from the start, which means making changes down the line becomes increasingly difficult and costly; such a style of development was appropriated from other industries before more suitable methodologies of software development were formalised. Rapid Application Development allows for a versatile development-driven

approach which naturally incorporates user feedback, design and testing as part of the cyclic process, which hopefully will result in a better, well-rounded final product.

4 Implementation

4.1 Technologies

Website and application created with HTML, CSS and Javascript with OFCP game page created dynamically with jinja2 templates.

Python backend handling dealing of cards, processing game state, AI's card placements and scoring of game board

Pure python networking with cherrypy, which is efficient and can handle up to 1000 concurrent requests which is more than enough for the requirements of this project.

Site hosted using DigitialOcean VPS running Ubuntu 14.04 with CherryPy behind nginx reverse-proxy[7]. This allowed for a more robust system architecture with reduced stress on the CherryPy server as nginx was configured to handle client requests, meaning that scalability in the future is more feasible.

Game states stored in database using MongoDB (which is a document-oriented database as opposed to a traditional relational database which decreases development time and reduces complexity as there is no need to constantly transform the data when reading from the database into the python backend. Mongodb is scalable and high performance, and allows for flexible data structures for example with optional values being handled trivially, with the databases getting type information from the data itself meaning they can map easily into program objects, which is specifically advantageous in this application because of the use of dictionaries to store the game state. The flexibility and ease of deployment of such a style of database makes their use well suited for web-applications such as this one) run in docker virtual environment.

Version control with github: OFCP-AI private repository. Use of version control is paramount as it allows for undesirable changes to be rolled back easily, and if something goes wrong there is always working versions available to roll back to. Using multiple branches (master and experimental) meant that a stable version could be maintained while new features were implemented safely on the experimental branch without affecting or potentially breaking the master version. Also allows for 'releases' for different versions of the software e.g. legacy client-side only, current with feature-full self-contained application

4.2 Intelligent Agent- algorithms, heuristics etc

write it all here - monte carlo, descision rules for placements/ pruning, EV. Everything. Wow. Do it. Soon.

First 5 placements approach - find all permutations of possible placements (13 choose 5 = ~1.2k) then remove duplicates (e.g. ADAHAKASKD and KDASAHASAC - different order but effectively same state) -> reduces max states to consider to ~240. Next even more pruning/ optimisation by implementing heuristics to evaluate hand and remove some obviously bad placements e.g. if AI is dealt two cards of same rank, will always place these together (removes states with these cards in separate rows). Also applies to trips, four of a kind etc. -> better the hand the less states considered. Worst case scenario with 5 unique ranked cards no straight/ flush potential = over 200 states, potentially suboptimal placements due to need to return move decision in <= 5 seconds (very little time for each simulation - 5 seconds to simulate 200 states will not produce best results except with exceptional hardware).

Subsequent card placements a lot easier - carry out simulations for as many seconds as possible (hard-coded limits can be tweaked in response to user feedback [sweet spot 2-4s, 5s meets requirements but makes game feel somewhat slow]). Aggregates EV for placements in each row and choose highest result, update state return placement info to frontend along with player's next card. With only 3 choices to make (place card in top, middle or bottom) can potentially get thousands of simulations for each placement which should give close to optimal results - see limitations in reflection e.g. potential for overvaluing moves. Simulated

[7] This tutorial was incredibly helpful for this purpose: https://www.digitalocean.com/community/tutorials/how-to-deploy-python-wsgi-applications-using-a-cherrypy-web-server-behind-nginx

random population of rest of board for scoring EV is naive but works well most of the time.

4.3 Live Implementation

The Application is available and playable at the following URI: http://www.alastairkerr.co.uk/ofc

5 Testing

usability tests - table of what tests, what happened
questionnaire and results, general evaluation

5.1 Unit Tests

continuous integration/ deploymentwith continuous integration when you make a commit to github it automatically connects to server and runs the tests, if they pass it deploys it http://code.tutsplus.com/tutorials/setting-up-continuous-integration-continuous-deployment-with-jenkins–cms-21511

Modular testing of code e.g. individual functions using unittest python framework

```
if (type(hand) is not str):
    print "Invalid hand (required type = string), " + str(hand) + " is " + str(type(hand)) + "\n"
    return None

if (len(hand) != 15):
    print "Invalid Hand. Required format e.g: c09c10c11c12c13 (clubs straight flush 9->King).\n"
    return None

try:
    print ('Reading in hand: ' + str(hand) + '. Reformatting now...\n' )
    cards_list = []
    formatted_hand = ""

    rank_dic = {'10':'T', '11':'J', '12':'Q', '13':'K', '14':'A'}

    for i in xrange(0,15,3):            # decode string to get each card name. index 0 -> 14 step 3
        suit = hand[i]
        rank_p1 = hand[i+1]
        rank_p2 = hand[i+2]

        suit = suit.upper()             # evaluator needs suit as uppercase char

        if suit not in ('H','D','S','C'):
            print "Invalid suit! Expected H, D, S or C. Actual:", suit
            return None

        rank = int(rank_p1 + rank_p2)   # get numerical value for rank
        if ( rank < 1 or rank > 14):
            print "Invalid rank. Accepted range 1-14.\n"
            return None
```

Fig 4.1.1 - Sample code from function 'reformat_hand_xyy_yx' in helpers.py script: use of input validation and try except blocks to catch errors

```
test_items = ( 'c05c06c07c08c09','s05c05h09s08d13','h13c01s03d05c07','invalid',100,'fakestring',('i','am','invalid'),'1234567891123345' )
for item in test_items:
    format_resp = helpers.reformat_hand_xyy_yx(item)
    if format_resp != None:
        print 'Formatted ' + str(item) + ' -> ' + str(format_resp) + '\n'
```

Fig 4.1.2 - Test inputs to ensure function works as intended

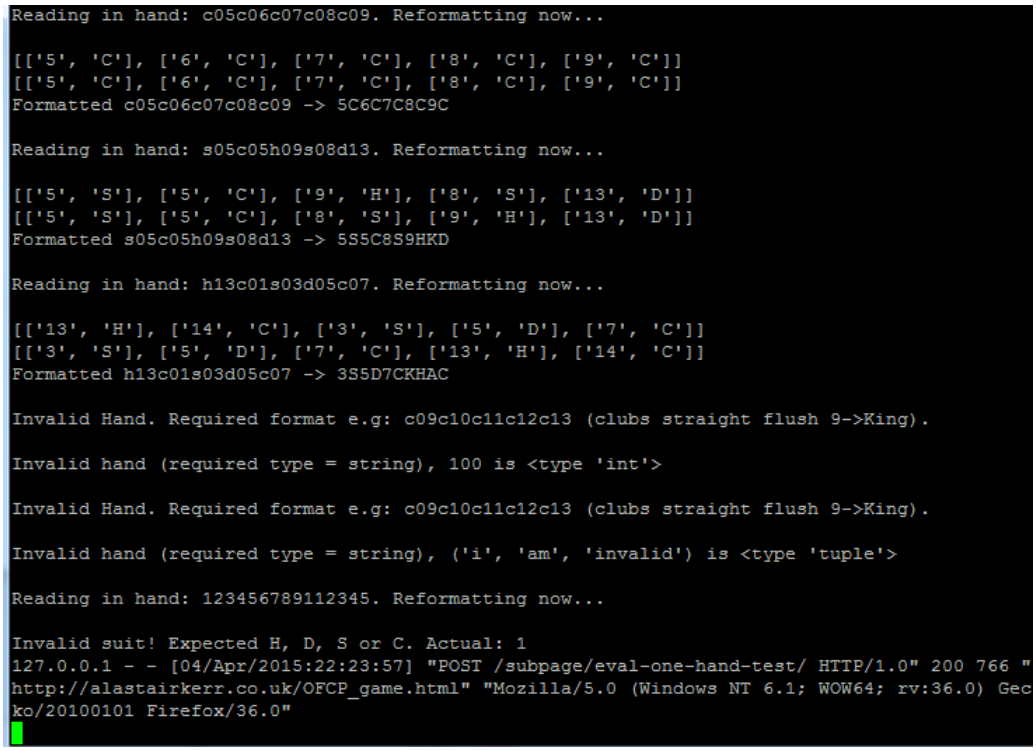


Fig 4.1.3 - Output – invalid hands are handled properly, throw exceptions/ print usage messages rather than throwing errors

5.2 Performance of AI versus human players

Alpha testing: playing individual games with participants vs AI Playing vs experienced players, new players – get an indication of AI's comparative skill level

5.3 Performance of AI versus other AI

Pit this intelligent agent vs other AI and/or previous/ alternative versions of itself. E.g. performance of AI with MCTS vs AI using AB pruning/ minimax, MCTS vs totally random placement: if AI is working well should vastly outperform a naive AI. Visualisations of performance e.g. graphs, tables of win rates Database storing moves -> this would allow for analysis of individual rounds, games etc.

^ 'probably' no time to do this. Explain time constraints, say this is what i would do.

6 Evaluation

6.1 Aims and Objectives

To what extent were the aims met? Sophistication and performance of AI? Were all features implemented?

Performance of website: http://tools.pingdom.com/fpt/#!/cXxmxY/http://alastairkerr.co.uk/ofc/play/5540d2e7b878ce0

6.2 Reflection

Reflection on project, decisions, performance etc.

Design - frontend is functional but could have been designed better from the start. e.g. originally hard-coding player's card image objects rather than dynamically creating them with javascript

Backend works well but if different technologies and languages were used could be more efficient - e.g. hand evaluators written in C using bitwise operators could evaluate millions of hands per second rather than hundreds of thousands - could shave off seconds of processing time which could either lead to increased responsiveness or allow for more games to be simulated by the AI making it more likely to find optimal solutions for hand placements.

The choice of a flexible software methodology worked well overall because of the evolving requirements and design choices, as well as the individual nature of the project. In comparison, in a large team of developers issues with this approach could arise from lacking a clear design focus and having limited control - a necessary trade-off that is an inevitable consequence of the increased flexibility this methodology enables. One important pitfall to avoid with Rapid Application Development is focusing too much on individual components without getting a clear view of the system's design, making minor changes without considering possibilities for an improved design structure. While there is generally a clear design phase before entering the initial implementation stage, there can be a tendency to omit a renewed design phase in subsequent implementation cycles leading to a lack of documentation which can have large consequences down the line; as Gerber et al. (2007) state in the analysis of one case study "... due to the fact that the design was not formally documented and reviewed, the discrepancy was only discovered after the implementation phase. This situation caused conflict between developers and analysts and in the end necessitated a redesign effort which put unnecessary pressure due to time constraints and limited resources on the whole development team". Design choices in early prototypes had a carry-on effect which meant that later down the line code refactoring was necessary in order to create a more coherent system structure, which potentially could have been avoided or reduced with a stricter design philosophy.

An apt example of this is seen in legacy prototypes of the application, which were client-side only. This was a choice that was made in order to quickly create a functional prototype, using JavaScript to simulate processes that would be handled elsewhere in the final application's architecture (such as dealing cards). This was useful because it resulted in a functional application which implemented some of the planned features, leading to a clearer understanding of the needs of the project, but had to be adapted later in order to create a more logical system structure which could meet the requirements, such as backend processing to handle the game states and calculate the Intelligent Agent's moves.

6.3 Improvements

add a new game button!

What can be done to improve the application/ AI in the future?

Limitations: "With any method based on random simulation, it is inevitable that poor quality moves will be chosen with nonzero probability, due to a particularly lucky run of simulations making the move appear better than it is. " - http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/viewFile/7369/7595 page 5

Due to the need for a compromise between finding the optimal solution and finishing the request in a reasonable amount of time the number of simulated games is limited and therefore it is possible that sub-optimal plays will be over-valued due to the element of randomness.

Improve frontend – make the app more visually appealing

Multiplayer support for players vs players as well as players vs AI(s) or players vs players vs AI(s)

Add support for other variants of OFC such as pineapple - includes elements of hidden information as players choose to discard one card, the identity of which is known only to them. Will increase complexity of game requiring further modifications/ improvements to Agent, and would make any performance issues even more of an obstacle.

The Intelligent Agent generally favours optimal card placements, but because of the element of randomness in evaluating moves there is the potential for the AI to mistakenly over-value a sub-optimal move. As Whitehouse et al. (2013, p5) state "With any method based on random simulation, it is inevitable that poor quality moves will be chosen with nonzero probability, due to a particularly lucky run of simulations making the move appear better than it is". This potential for inaccurate evaluation of a move's strength has an inverse relationship with the number of iterations of simulated games - as the number of iterations increase the result diverges to the optimal solution, meaning that with an infinite amount of simulations the probability of finding the best move is 1. The performance of the Intelligent Agent therefore could

be improved with more processing power and/or a longer allocated time to simulate games, although it is important to note that there are diminishing returns with this strategy; doubling the iterations does not mean that the results will be twice as good.

This ties in with the choices made for the implementation and configuration of the Agent, specifically in regards to compromises between responsiveness and finding the best move - in the specified requirements the Agent was intended to take no more than 5 seconds to calculate its turn, and the application satisfies this requirement. However, increasing the iterations would make satisfying this criteria infeasible without increased processing power or through further optimisation. This could be achieved in various ways, for example by rewriting the application in a more efficient language such as C, or by implementing more advanced heuristics to reduce the complexity of the calculations, or using a different algorithm such as a more advanced implementation of the Monte Carlo method like Monte Carlo Tree Search with UCT, as discussed in Section 2 of this dissertation. Overall the implementation of the Intelligent Agent satisfies the specified requirements and works well for its intended purpose in the scope of this application, but for larger scale implementations would likely need to make use of one or more of these changes in order to achieve increased scalability, for example for use in a commercial application with thousands of concurrent users.

7 Conclusion

In conclusion the Intelligent Agent produced performed adequately well and largely ticked the boxes in terms of requirements specified, demonstrating the versatility of Monte Carlo methods; with little domain specific knowledge, usable results were obtainable. The Web-Application itself worked perfectly for its intended purpose and shows promising scalability, with a robust and sensible system architecture, and the final interface produced was user-friendly and visually appealing. The technologies and methodologies employed enabled very fast turnover of prototypes and new software iterations, which turned out to be critically useful in making improvements to the system in response to user feedback.

Room for improvement can be had in many aspects however. Using Python allowed for quick development but the application could have benefited from the use of a more efficient language such as C, although these performance concerns were partially mitigated through the use of PyPy. The Intelligent Agent's simplistic approach could be expanded upon and improved, and fundamentally there are limitations with the approach used. The simple Monte Carlo methods used produce good results but are CPU intensive and necessarily require a tradeoff at some level between finding an optimal solution and getting fast results. It would be interesting to note the effect and improvements that implementation of more sophisticated Monte Carlo methods such as MCTS with UCT could achieve, and I look forward to further development of the Intelligent Agent in this respect.

8 Glossary of Terms

8.1 Poker Variants

- **Texas Hold'Em** is a popular variant of Poker where each player receives 2 cards for use individually in combination with 5 community cards shared between all players, with players combining any of their available cards in order to create the strongest standard poker hand possible
- **Chinese Poker** is a variant of poker where players are dealt 13 cards which they must arrange into three rows, placed face down. Players announce in clockwise order whether or not they wish to play their hand, and then all players announce their royalties and show their cards
- **Open Face Chinese Poker** is a variant of Chinese Poker where players act in clockwise order, receiving first 5 cards which are placed face up and then one card at a time until all players have placed 13 cards. Players must create valid hands consisting of stronger poker hands in lower rows, and score points from their opponents for winning corresponding rows. Additional points known as royalties can be won for particularly strong hands
- **Pineapple OFC** is a sub-variant of Open Face Chinese Poker, following the same basic rules as the standard variant with the distinction that in subsequent rounds after the initial 5 card placements,

players receive 3 cards and choose to place 2 and discard 1. This introduces higher action play as well as elements of hidden information as other players are unaware of which cards their opponents have discarded, although there is potential to infer this information based off of how the player acts

8.2 Poker Hands Guide (Weakest to Strongest)

- **High Card** is the lowest ranking poker hand and is the default when no other hands have been made. An example High Card is the Jack of Spades, which would beat any hand comprised of a High Card ranked 10 or less, but would lose to a High Card Queen, King, Ace, or any stronger poker hand.
- **Pair** is the second weakest poker hand. A player has a pair when they have two cards of the same rank, for example the 7 of Hearts and 7 and Clubs would form a Pair of 7s.
- **Two Pair** is the next strongest hand rank, consisting of two different pairs. For example having the 5 of Spades, 5 of Diamonds, 9 of Hearts and 9 of Clubs would form Two Pair 9s and 5s. When comparing a Two Pair to another Two Pair the highest ranked pair takes precedence.
- **Three of a Kind** or a **Set** beats Two Pairs, Pairs and High Cards and consists of three same-ranked cards, such as the Ace of Spades, Ace of Diamonds and Ace of Hearts, which would form Three of a Kind Aces
- **Straight** is a hand where a player has 5 sequential cards, such as 4 of Diamonds, 5 of Hearts, 6 of Hearts, 7 of Clubs, 8 of Spades, which would form a Straight 8 High. The lowest ranked straight spans Ace to 5 and the highest ranked straight spans 10 to Ace. It is important to note that straights do not wrap around; you cannot form a straight such as Queen, King, Ace, Deuce, 3.
- **Flush** is one of the stronger poker hands, consisting of cards which are all the same suit. For example the 6 of Hearts, 9 of Hearts, Jack of Hearts, Queen of Hearts and King of Hearts would form a King High Flush.
- **Full House**, sometimes known as a **Boat**, is one of the strongest poker hands available, and comprises both a Three of a Kind and an additional Pair. For example Three of a Kind Tens with Pair of Jacks would combine to form a Full House, Tens full of Jacks.
- **Four of a Kind** is a hand obtained when a player has every instance of a particular card rank, such as King of Hearts, King of Diamonds, King of Clubs and King of Spades which would form Four of a Kind Kings.
- **Straight Flush** is effectively the strongest poker hand possible, consisting of 5 sequential same suited cards. For example the 4 of Spades, 5 of Spades, 6 of Spades, 7 of Spades and 8 of Spades would form a Straight Flush 8 High.
- **Royal Flush** is a special instance of a Straight Flush where a player has the 10, Jack, Queen, King and Ace of a particular suit. Royal Flushes are particularly rare; in Texas Hold'Em the probability of getting a Royal Flush is approximately 0.000154%.

8.3 Open Face Chinese Poker Terminology

- A **row** is a set structure for placing cards. At the end of a game when scoring occurs poker hands in each player's rows are compared to the opposing player's hand in their corresponding row. There are three different rows as described below
- **Bottom Row** or **Back Hand** is the foundation row, and consists of 5 cards. Out of all three rows this must have the strongest poker hand or the player fouls.
- **Middle Row** or **Middle Hand** also consists of 5 cards. It must have a weaker hand than Bottom Row in order for the player's hand to be valid.

- **Top Row** or **Front Hand** consists of 3 cards, meaning the best possible hand here is a Three of a Kind (in most variants 3 card straights and flushes do not count). Top Row must have a weaker hand than both middle and bottom row.
- **Scoop** is a bonus awarded to a player when they win all 3 rows against an opponent. On top of the standard +1 point per row won, the player is granted an additional bonus of +3 points which is also taken from their opponent.
- **Fouling** occurs when a player plays an invalid hand, for example by putting a stronger hand in their middle row than their bottom row. When a player fouls any of their royalties are null, and their opponent is automatically awarded a scoop bonus so long as their hand is valid.
- **Kicker** is the term used to describe the next card taken into account when comparing two otherwise equal hands. For example, if both Player 1 and Player 2 have a Pair of 8s then the rest of their cards would be considered, and whichever player has the highest rank wins. If both players kicker is equivalent then the next highest kicker will come into play and so on.
- **Royalties** are bonus points awarded to player's for particularly strong hands. Just like any other points a player wins, they are taken directly from opposing players. Hands in higher rows score higher royalties than equivalent hands in lower rows. For example, a Full House in bottom row is worth 6 points in bottom or 12 points in middle. Another example is that a Three of a Kind in bottom row scores no royalty, but gets 2 bonus points in middle and between 10 and 22 points in top depending on the rank (Three of a Kind Deuces scores 10 points up to Three of a Kind Aces with 22 points). See here for a full list of royalties: http://www.wsop.com/2013/Open_Face_Chinese_Structure_Sheet.pdf

8.4 Computer Science and Mathematical Terminology

- **Algorithms** are precisely defined step-by-step instructions describing a set of operations to be performed
- **Artificial Intelligence** is a field of study in Computer Science which focuses on simulating intelligent behaviour in computers.
- **Automated Planning and Scheduling** is a branch of artificial intelligence which focuses on strategies and actions to be performed by Intelligent Agents or autonomous robots. Research into optimisations and improved heuristics is a key area of interest due to the potentially huge complexity of problems and solutions.
- The **Branching Factor** is the average amount of branches from a node in a tree, indicating the tree's complexity. At a depth of d with a branching factor of B there would be approximately B^d nodes.
- **Combinatorial Explosion** is the phenomenon of exponential growth encountered in search problems. Deeper searches become increasingly complex multiplying by the branching factor at each level
- A **Game Tree** represents possible states in a game, with each node representing a possible position and each edge representing a possible move
- **Game Theory** is a branch of mathematics focused on determining optimal strategies and decisions in competitive situations
- **Heuristics** are techniques used when classical approaches would fail or be prohibitively slow to complete. Heuristics use intelligent guesses to reduce the complexity of problems. For example in the context of a Chess game rather than trying to explore every single possible move and subsequent tree branch, a heuristic based approach would ignore branches starting with clearly bad moves
- **Hidden Information** is relevant information available to one or more agents but not others. For example in a Poker game, the cards that players individually possess or have discarded are known to them but are hidden from other players.

- An **Intelligent Agent** is an autonomous software entity that perceives and acts upon its environment, performing reasoning in order to solve problems and determine actions, exhibiting goal-oriented behaviour.
- The **Optimal Move** or **Optimal Strategy** is the move or strategy that will lead to the most favourable outcome for an Agent
- **Perfect Information Games** are games where at any stage all relevant information is available to an agent in order to inform its decision. In comparison in an **Imperfect Information Game** certain information about the game state or prior actions are unknown.
- **Rule Based Systems** store expert knowledge for a particular domain as a set of rules which can be used in various artificial intelligence applications. Rule Based Systems are used extensively for a wide range of purposes such as game playing, credit card authorisation and fraud detection.
- **Solved Games** are games where the outcome (Win/Loss/Draw) can be predicted at any stage assuming optimal play by all players
- A **Zero-Sum Game** is a game in which a player's gains are losses are balanced by another player's gains or losses. Open Face Chinese Poker is an example of a Zero-Sum Game as a player's gains are directly taken from their opponent.

9 Bibliography

- Mandziuk, J., 2010, *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*, Springer-Verlag
- Shannon, C., 1950, *Programming a Computer for Playing Chess*, Philosophical Magazine, Ser .7, Vol. 41, No. 314
- Özcan, E. and Hulagu, B., 2004, *A Simple Intelligent Agent for Playing Abalone Game: ABLA*, Proc. of the 13th Turkish Symposium on Artificial Intelligence and Neural Networks, pp. 281-290
- Von Neumann, J, 1928, *Zur Theorie der Gesellschaftsspiele,* Math. Annalen. 100, 295-320
- Von Neumann, J. and Morgenstern, O., 1944, *Theory of Games and Economic Behavior*, Princeton University Press
- Casti, J., 1996, *Five golden rules: great theories of 20th-century mathematics – and why they matter*, New York: Wiley-Interscience
- Kjeldsen, T., 2001, *John von Neumann's Conception of the Minimax Theorem: A Journey Through Different Mathematical Contexts,* Springer-Verlag
- Samuel, A., 1959, *Some Studies in Machine Learning Using the Game of Checkers*, IBM Journal ,Vol 3
- Dizman, D., n.d., *A Survey on Artificial Intelligence in Stochastic Games of Imperfect Information: Poker*
- Russell, S. and Norvig, P., 2003, *Artificial Intelligence: A Modern Approach*, Upper Saddle River, New Jersey: Prentice Hall, 2nd Ed
- Eckhardt, R., 1987, *Stan Ulam, John Von Neumann, and the Monte Carlo* Method, Los Alamos Science Special Issue 1987, p131-143
- Coulom, R., 2007, *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*, Computers and Games, 5th International Conference, Springer
- Chaslot, G. et al, 2008, *Progressive Strategies for Monte-Carlo Tree Search*, New Mathematics and Natural Computation, p343-357

- Browne, C. Et al, 2012, *A Survey of Monte Carlo Tree Search Methods*, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1.
- Kocsis, L. and Szepesvári, C., 2006, *Bandit based Monte-Carlo Planning*, Computer and Automation Research Institute of the Hungarian Academy of Sciences
- Cowling, P. Et al, 2012, *Information Set Monte Carlo Tree Search*, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 2.
- Whitehouse, D. Et al, 2013, *Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game*, University of York
- Nielsen, J., 1994, *Usability Engineering*, San Diego: Academic Press
- http://scrambledeggsontoast.github.io/2014/06/26/artificial-intelligence-ofcp/ - Intelligent Agent 'Kachushi' for Open Face Chinese Poker using Monte Carlo methods, implemented in Haskell, accessed 2015
- Gerber, A. Et al, 2007, *Implications of Rapid Development Methodologies*, CSITEd 2007, p242-243